# Introduction

The adapter pattern is a structural design pattern that allows objects with incompatible interfaces to collaborate. It acts as a bridge between two classes by providing an alternative interface to an existing class.

The focus of this exercise is on demonstrating the concept and implementation of the adapter pattern. While a fully functional application is not required, the provided code should be compilable without syntax errors and effectively illustrate the pattern's structure.

# Question 1:

You're working on a project that requires the integration of two separate systems. The interface used by one system is incompatible with the interface used by the other system. In the first system, the method foo() takes an integer as a parameter and returns a string. In the second system, the method bar() takes a text as a parameter and returns an integer. How would you use the Adapter pattern to allow apps to use the second system while continuing to use the first system's interface? Make a Python program that shows how you would use the Adapter design to tackle this problem.

---

# Question 2:

Implementing an RectangleAdapter

- Consider the following classes: LegacyRectangle and ModernShape. The LegacyRectangle class depicts a rectangle by having width and height attributes, but the ModernShape class has area and perimeter attributes. Both classes have distinct interfaces that make them incompatible.
- Create a RectangleAdapter adapter class that implements the ModernShape interface. The LegacyRectangle object should be adapted to the ModernShape interface using the RectangleAdapter. Also, ask yourself which class is the adaptee in this case?
- Using the attributes of the LegacyRectangle object, implement the RectangleAdapter class's get_area() and get_perimeter() methods.
- Create a ShapeAdapter adapter class that implements the LegacyRectangle interface. The ModernShape object should be adapted to the LegacyRectangle interface by the ShapeAdapter. Also, ask yourself which class is the adaptee in this case?.
- Using the attributes of the ModernShape object, implement the ShapeAdapter class's get_width() and get_height() methods.

---

# Question 3:

Consider the class LegacyPaymentGateway to represent a legacy payment processing system. This class only accepts credit card payments and exposes the following methods:

```
class LegacyPaymentGateway:
    def process_credit_card_payment(self, credit_card_number: str, expiration_date: str, cvv: str, amount: float) -> bool:
        print("Simulates processing a credit card payment")
```

```
    return True;
```

You've been assigned to merge this old system with a new payment processing system that requires PayPal payments. The following interface is exposed by the new payment processing system:

```
class PayPalPaymentGateway(ABC):
    @abstractmethod
    def process_paypal_payment(self, email_address: str, amount: float) -> bool:
        pass
```

Your goal is to design an adapter class that combines the LegacyPaymentGateway interface with the PayPalPaymentGateway interface. This adaptor should allow customers to make PayPal payments without having to interface directly with the LegacyPaymentGateway class.

This adapter class connects the legacy payment gateway to the new PayPal payment processing system. It enables clients to make PayPal payments without having to interact with the incompatible interface of the legacy system.

---

## Question 4:

Consider an outdated notification system that can only send notifications via SMS. You wish to combine this old system with a new notification system that can send notifications via email, push notifications, and social media updates. To combine the legacy notification system with the new notification system, create an adapter class.

The legacy notification system exposes the following method:

```
class LegacyNotificationSystem:
    def send_sms_notification(self, phoneNumber: str, message: str) -> None:
        print("Simulates sending an SMS notification")
```

The new notification system exposes the following interface:

```
class NewNotificationSystem(ABC):
    @abstractmethod
    def send_email_notification(self, emailAddress: str, subject: str, body: str) -> None:
        pass

    @abstractmethod
    def sendPushNotification(self, deviceToken: str, title: str, message: str) -> None:
        pass

    @abstractmethod
    def send_social_media_update(self, social_media_platform: str, postContent: str) -> None:
        pass
```

Your goal is to write an adapter class that allows clients to send messages utilizing the new notification system without interacting with the LegacyNotificationSystem class directly. The adaptor should be able to transform SMS notifications into notifications appropriate for the new system.